

第3章

システム全体を見渡しながら回路設計を行う

――IPコア，インターフェース回路，外部デバイスをつないで実機動作させる

森岡澄夫

高性能なIPコアを設計し，RTLコードを作成して論理合成しても，それだけではシミュレーションしかできません．最終的に実チップ上で動かなければ回路は役に立ちませんが，そのためにはデータ入出力などを行う周辺回路や制御ソフトウェアがかならず必要になります．たとえ簡単な構成でも，周辺回路やソフトウェアまで合わせた全システムを一度自作してみると，見渡せる世界は大きく広がります．本稿では，パソコン用の画像処理システムの作成を例に，LSI設計を含むシステム開発の進めかたを説明します．なお，本稿で紹介する回路の設計データは本誌のWebサイト（http://www.cqpub.co.jp/dwm/）からダウンロードできます．（著者）

筆者は，本特集向けにパソコン用の画像処理システムを作成しました（その経緯は，pp.84-90のAppendix「実録『画像処理システムができるまで』」を参照）．その際，机上で回路を設計してシミュレーションするだけではなく，パソコンとつないでFPGAボード上で実際に動かしてみました．筆者は，こうした実機による動作確認をできるだけ行うことをお勧めします．その理由を以下に簡単にまとめます．

● シミュレーションで動いても実機が動くとは限らない

一つ目の理由は，回路設計およびシステム設計において「シミュレーションと実機動作は異なる場合がある」ということを理解するためです．例えば，シミュレーションで動かないRTL（register transfer level；現在のハードウェア記述言語を用いた回路設計では，このレベルの記述を使用することが多い）コードを論理合成したところで，実機ではやはり動きません．逆に，シミュレーションではRTLコードが一見正しく動いていたとしても，実機が動くとは限りません．言うまでもなく，実機で動かないRTLコードには何の価値もありません．また，そのようにシミュレーションと実機がかい離する傾向は，今後Cベース設計の導入などで設計の抽象度が上がれば，より顕著になっていきます．

それではシミュレーションと実機では何が違っていて，どうすれば実機が動くように設計できるのでしょうか？ また，シミュレーション結果をどのようにとらえていくべきなのでしょうか？

それを知るには，雑誌や書籍の解説記事をたくさん読んで知識を詰め込むよりは，一度手を動かして実機を作ってみるほうがずっと効果的だと筆者は考えます（そう書くと

Column　回路設計の「ココがすてき」

筆者は子どものときから電子工作が好きで，それをそのまま職業にしました．ですから，しごとでミスをして自分の未熟さを痛感したり，思うように業務が進まないことがあったりしても，しんそこ苦痛に思ったことは一度もありません．「回路設計」というしごとの魅力は，自分の作ったものが世の中に出て人の役に立つとか，進歩の速い分野なので刺激があって楽しいということが挙げられます．しかし，いちばん根っこにあるのは「自分の作ったものがうまく動くのを見るのが単純にうれしい．それが複雑で凝ったものであればなおさら」という達成感です．これはソフトウェアやメカなど，もの作りに携わる人みんなに共通しているのではないかと思います．

KeyWord　Stratix，RTL設計，API関数，PCIバス，SDRAM，シンクロナイザ，セットアップ時間，ホールド時間，ハンドシェイク，シミュレーション

本稿の意義があやしくなるが…）．実機作成はさまざまな機材および準備が必要なうえ，何から手をつけてよいかわかりにくく，最初はしり込みしがちです．しかし，実際にやってみると机上でのRTL設計よりもはるかに楽しいものです．もし，たまたまうまく動かせなかったとしても，それはそれで良い経験になります．

● システムを作ってみると回路設計の考えかたが変わる

二つ目の理由は，メインの回路ばかりではなくその周辺回路やソフトウェアにまで目を向けることで，メインの回路の設計のしかたもグッと良くなるからです．システム全体が見えてくれば，次のような著しくバランスを欠いた設計（ちょっと極端なたとえだが…）を行うことが少なくなります．

- パソコンとFPGAボードの間の接続にPCIバスを使うシステムで，データ転送速度が1Gbps未満（33MHz時）であるにもかかわらず，FPGAに100Gbpsの高性能で巨大な回路を載せてしまう．→その性能を発揮させることは決してできない
- データを1バイトごとFPGAに送っては戻すといった処理を行い，ハードウェア性能を殺してしまう．→オーバヘッドが減るように，データ転送はなるべくまとめて行うほうがよい．
- ちょっとしたデータ変換や1回しか行わない浮動小数点演算など，ソフトウェアで簡単に作れる処理をすべてハードウェアで実現しようとする．→回路がいたずらに複雑になるうえ，デバッグも難しくなる．

● 複数人による共同開発をうまく進めるには

三つ目の理由はより実務的なことですが，システム全体を見渡して動くモノをしあげるスキルそのものに価値があるからです．

しごとでもの作りを行う場合，複数人（場合によっては複数社）で作業を分担するのが普通です．そうした場合，それぞれの担当箇所や開発スケジュールなどを決めてしごとをうまく進めていくために，システム全体を把握できる人が必要です．もちろんリーダ役の人だけでなく，実務担当者も意識して全体をながめるようにしていないと，ほかの担当者との技術的な調整を図れませんし，問題が起きたときに解決できない状況になります注1．

1 画像処理システムの機能と構成

今回作成したものは，比較的実用に近いレベルのシステムです注2．画像処理回路をFPGAボードに搭載し，パソコン上で動く一つの画像処理システムの形にまとめています．画像処理回路本体の設計については，本特集の第4章を参照してください．ここでは，画像処理回路を使って一つのまとまったシステムをくみ上げる部分を説明します．

● 作るのは画像処理のアクセラレータ・ボード

図1に本システムの概要を示します．パソコン上にある24ビット・ビットマップ・ファイルのデータをFPGAボードに転送し注3，そのうえでフィルタなどの画像処理を施し，再びパソコンへ戻してビットマップ・ファイルで結果を得ます．一連の操作は，筆者が作成した**図2**に示すようなGUI（graphical user interface）を使ってパソコン上で行えます．OSはWindows 2000/XPに対応しています．実行可能な画像処理の一覧を**表1**に示します．GUI上でパラメータを設定して多種多様な変換操作を行えますが，ここでは詳細は割愛します．第4章の記事や配布データ（本誌のWebサイトからダウンロード可能）の中の説明を参照してください．

FPGAボードとして，今回はCQ出版社の「Stratix評価キット（米国Altera社のFPGA『EP1S10』を搭載）」(2)を使用しました．本評価キットでなくても8Mバイト以上のSDRAMとPCI端子を搭載しているFPGAボードであれば，ピン割り当てなどの軽微な変更で今回のシステムは動くと思います．FPGAボードは，パソコンの33MHz PCIスロットに挿して使います．パソコンとしてはなるべくハードウェア・リセット・スイッチが付いているものを使用することをお勧めします．また，FPGAプログラミング用のパソコンは別途用意しておきます注4．

注1：一見あたりまえのことなのだが，現実にはなかなか思いどおりにいかないことが多いと思う．うまくいかない開発では，「自分のかかわったところだけ動けばいい」というムードが広がっていることが多いようだ．

注2：できれば本誌2005年3月号（参考文献（1））を併読していただきたい．「メインの回路だけでなく，その周辺回路やソフトウェアまで合わせたシステムを作る」というテーマで，例題を交えながらできるだけ簡潔に説明している．

注3：最大サイズの画像をFPGAボードへ転送するのに約1秒（4Mバイト/s～8Mバイト/s），FPGA上の画像処理に約0.5秒かかる．

注4：パソコンの電源ON時にFPGAはプログラミングされていないが，OSが立ち上がった後でFPGAをプログラミングしてもPCIデバイスとして認識されない場合が多い．PCIクロックが停止してしまうためである．そこで，ハードウェア・リセットを押しながらパソコンの電源を入れ，FPGAをプログラミングした後でリセット解除すると，うまく動作する．また，FPGA開発環境は別のパソコン上に作っておくこと．参考文献（3）に詳しく説明されている．

なお，今回はシステムをとにかく作ってみることが第一目的なので，機能や性能についてはあまり追求していません．しかし，次のようなくふうを施すことで，より実用品に近いシステムになるでしょう．

- 今回は1枚の画像を入力とする処理しかないが，重ね合わせやマッチングなど，2枚の画像を入力とする処理を作ってみる
- FPGAボードでの処理の終了をパソコンに通知するために，PCI経由の割り込みを使う．今回は処理時間が短いので，割り込みは使っていない
- FPGA内部回路の動作クロックを上げる．今回はPCIクロック×2の66MHzで動作させたが，まだ余裕がある

● 実機作成とIPコア設計の相違点

RTL設計（IPコアの設計）と実機作成において，もっとも異なるのは次の2点です（**表2**）．

- **ポイント1** ── 単一クロックで動くIPコアとは異なり，実機の場合は動作速度の異なるいくつかのブロックをつなぎ合わせるのが普通．ここでいうブロックとは，チップ内の回路モジュールだけでなく，チップの外にある各種デバイスやソフトウェアまで含まれる．それらの間でデータを確実にやりとりする方法が必要になる
- **ポイント2** ── 実機を動かしていく作業は，RTLシミュレーションの場合と比べて，ボトムアップの色合いが強いものになる．どの部分がどのように動いているかを自由に追跡できないからである．このため，設計

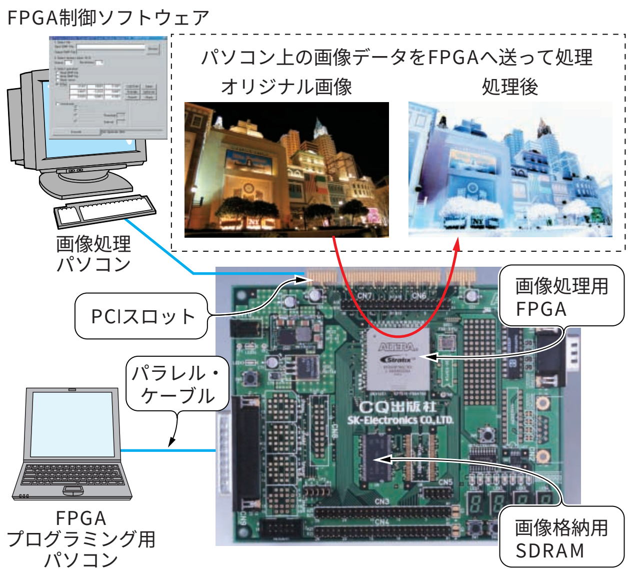


図1　画像処理システムの全体図

画像処理IPコアをFPGAに実装し，そのFPGAを搭載したボードをパソコンとPCIバス経由で接続する．パソコン上の画像ファイルをボードへ転送し，いろいろな加工処理を行った後，結果をパソコンへ戻す．この一連の処理操作は，パソコン上の制御ソフトウェア（GUI）から簡単に行える．FPGAのプログラミング（コンフィグレーション）には別のパソコンを用いる．

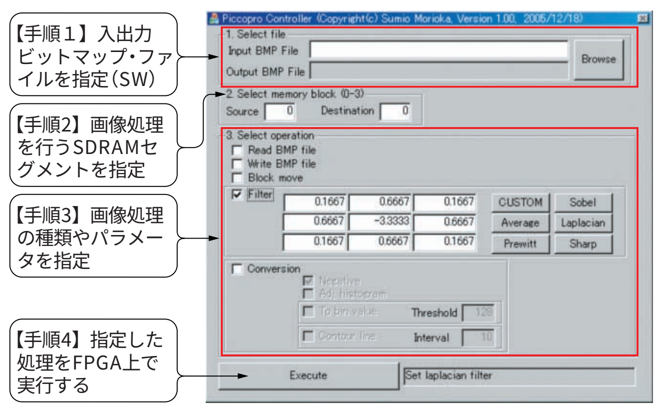


図2　画像処理システムの基本的な使いかた

図のようなGUIを用い，パソコン上からFPGAボードを簡単に使うことができる．画像データ・ファイルやパラメータ値，処理の種類などを入力してスタート・ボタンを押すと，必要な情報がFPGAへ送られて処理が行われる．

表1　作成する画像処理システムの基本機能

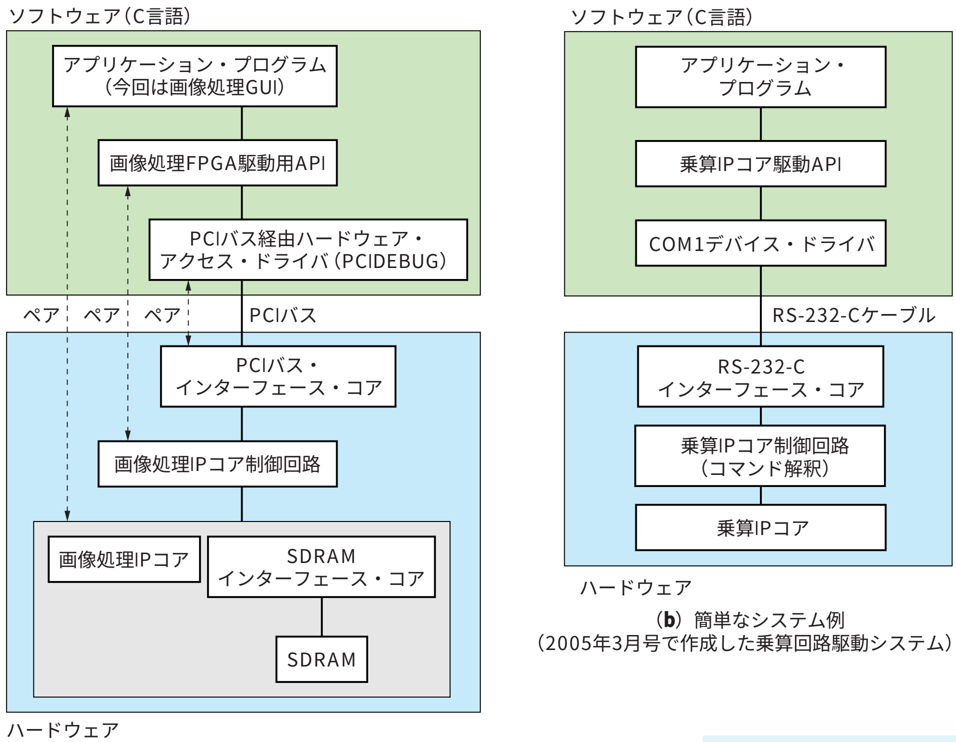
よく知られている代表的な画像変換処理を自由に行える．FPGA内部でさらに高速なクロックを生成すれば，処理速度をもっと向上させられるが，今回はそこまではやっていない．

入力画像形式	24ビットRGB画像（BMPファイルなど）
出力画像形式	24ビットRGB画像（BMPファイルなど）
画像サイズ	1×1～1024×512画素（可変）を計4枚
処　理	● 3×3フィルタ処理（係数は自由に設定可能） ● レベル変換（RGB値のテーブル置換） ● ヒストグラム算出，ヒストグラム平たん化
処理速度	実測約2.4Mピクセル/s，33MHz PCIクロック時（速度向上は可能）

表2　画像処理システム設計における技術上の注意点

実機で動かすシステムの設計とテストには，一般のRTL設計とは異なるセンスが必要である．とくに，ソフトウェアとハードウェア，あるいは複数のハードウェア・ブロックなど，実機では速度（クロック）の異なる複数の処理が並列に動くので，処理間の同期や回路接続部の設計にかなりの注意が必要となる．また，実機がシミュレーションどおりに動くとは限らないし，ボトムアップでテストを進める必要もある．

インターフェース回路設計	PCIバス・ターゲット，SDRAM
回路間接続	クロックの異なる回路の接続方法，全体回路のまとめかた
ソフトウェアとハードウェアの間のやり取り	レジスタ経由のデータ交換，ハードウェア処理の起動，ソフトウェアとハードウェアの同期（ポーリング，割り込み）
ソフトウェアのプログラミング	GUIのイベント・ドリブンとハードウェア制御処理との兼ね合い
開発作業のやりかた	実機を動かしていくためのテスト手順，設計モジュール分割のしかた



(a) 今回作成したシステム

(b) 簡単なシステム例 (2005年3月号で作成した乗算回路駆動システム)

図3
画像処理システムの基本設計
ソフトウェアとハードウェアを合わせたシステムを作るうえで重要なポイントの一つは，処理の階層化である．ソフトウェア側では，アプリケーションを最上位としてハードウェア側へ向かって階層を作り，抽象度を下げていく．ハードウェア側では，ソフトウェアの各階層とペアを作りながら，逆の形で抽象度を上げていく．各ブロックの作成は，高性能・高機能をねらわなければ意外と簡単に行える．学習目的であれば，初めから高性能をねらわず，まずは全体をしあげて実機として動かしてみることが大事．

モジュールの分割のしかたやテスト手順をくふうする必要がある

これら二つについては，後で重点的に説明します．まずは，本システムの全体構成をざっと見てみることにします．

● ソフトとハードをペアにした階層設計を行う

図3に示すとおり，ソフトウェア(今回はC言語を使う)からハードウェアに至るまでのデータの流れに沿って，処理ブロックの階層を作ります．階層を作るおもな理由は，意味的にまとまった処理ごとに区切りをつけることによって，設計とデバッグを容易にすることです．この階層はやみくもに作っているわけではなく，いくつかのルールに基づいて構成しています．具体的な内容は後述します．

- 階層の両端にあるのが，もともと実行したい処理(画像処理)を担当するブロック
- 中間にある階層はいずれもデータ転送の役割を担う．データ・フォーマット変換や速度変換などを行う
- ソフトウェアからハードウェアへ向けて，ソフトウェア側では徐々に抽象度が下がる．逆に，ハードウェア側では徐々に抽象度が上がる．ここで，抽象度とは「処理が本来の内容(画像処理)そのままであるほど高く，直接関係ない処理が多いほど低くなるもの」と考えていただきたい
- ソフトウェアの階層とハードウェアの階層でペアを作る

こうした階層の作りかたは，ほかのシステムの場合でも基本的に同じです．

今回は，ソフトウェア側とハードウェア側のそれぞれに3段の階層があります．各階層のおおよその機能は以下のとおりです．

1) 画像処理そのものを担当する階層

ソフトウェア側の最上段(図2のGUI)とハードウェア側の最下段(第4章で解説するIPコア)のペアは，フィルタなどの画像処理そのものを担当します．ユーザ側から見える「GUI経由でデータを送ると処理がなされる」という動きは，このペアによるものです．

2) 画像処理IPコアのAPIを提供する階層

システムの内部には，GUIが受け取ったデータをIPコアへ転送したり，IPコアに処理開始コマンドを送ったりする通信経路があります．つまり，GUIは直接FPGAを操作するのではなく，専用の通信経路を介して操作するのです．この経路を提供するのがAPI(application programming interface)という階層です．これもソフトウェア側とハードウェア側でペアを構成しています．

データを転送するというと簡単なようですが，GUIとIPコアの速度差を吸収したり，ソフトウェアとハードウェアのデータ・フォーマットを変換するなど，実際にはいろい

表3
アプリケーションに提供されるAPI関数の実例

FPGAで行う個々の画像処理が関数になっているだけである．関数を呼んでリターンしてきたとき，処理は完了している．ただし，時間のかかる処理をFPGAで行いたい場合は，プログラミングがやや複雑になる．その場合，割り込みによって処理完了通知を受け取る方式をとるとよい（とくにイベント・ドリブン処理のGUIを用いる場合など）．

API関数	内　容
piccopro_ipif_init( )	回路を初期化し，各種情報（デバイス・ハンドラに相当）を取得する
piccopro_ipif_setdata( )	処理するデータをFPGAボードへ転送する
piccopro_ipif_getdata( )	処理したデータをFPGAボードから取り出す
piccopro_ipif_resetsize( )	画像サイズを設定する
piccopro_ipif_execfilter( )	フィルタ処理を実行する
piccopro_ipif_exectblconv( )	テーブル変換処理を実行する
piccopro_ipif_exectrans( )	FPGAボード内で画像データのコピーを実行する
piccopro_ipif_gethist( )	画像のヒストグラムを取得する
piccopro_ipif_flattern_hist( )	画像のヒストグラム平たん化処理を行う

ろとやる事があります．

3）PCIバスによるデータ転送階層

API層の下には，ソフトウェアとハードウェアのちょうど接点にあたる部分として，PCIバスを実際に使ってデータ転送を行うための階層があります．**図3**ではこれを最下層としていますが，現実にはパソコンの内部にさらにいくつかの階層があります（今回はそこには触れない）．

この階層の処理を差し替えることで，例えばUSBなど，PCIバス以外のインターフェースで通信を行うようにすることも可能です．

2 階層間のデータ交信のしくみ

次に，上述した実機作成のポイントの一つ目である，各ブロック間のデータのやりとりについて，階層の上から順に説明します．

● アプリケーションとAPIのやりとりは普通のCと同じ

アプリケーション・プログラム（GUI）から使用できるAPI層（**図3**の上から2番目）のC関数を**表3**に示します．これらの関数は，次の二つに分けられます．

- FPGAボードを初期化し，デバイス・ハンドラに相当する値を取得する関数
- パソコンのメモリ上のバッファに格納された画像に対し，個々の画像処理を行う関数

アプリケーションから見た画像処理の動きを**図4**に示します．アプリケーションは実行したい処理の関数を呼ぶだけでよく，関数から戻ってきたときには処理が完了しています．このため，アプリケーションのプログラミングでは，処理がハードウェアで行われているということをほとんど意識しなくてすみます．ソフトウェアで同一名の関数を作った場合との違いは速度だけです．そのため，ハードウェアが何もでき上がっていない状況でも，ソフトウェアだけでアプリケーション部を作成，デバッグすることが可能です．

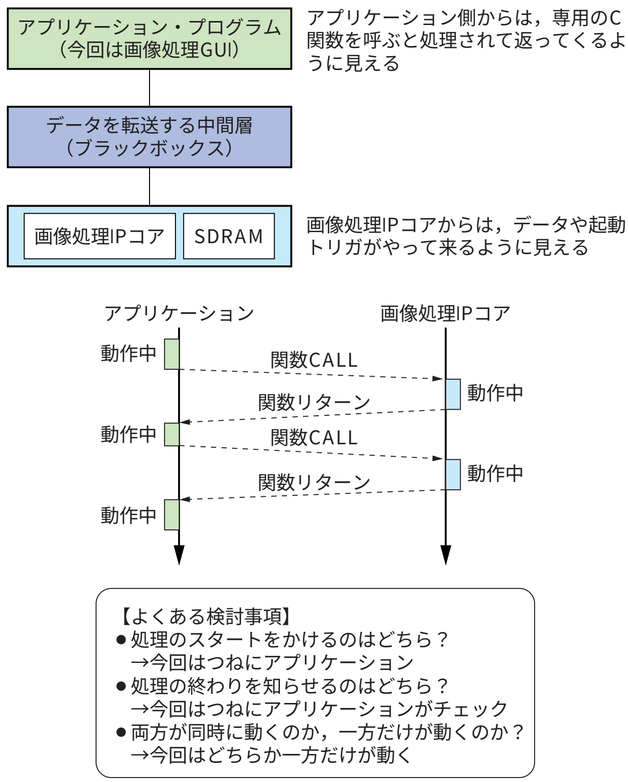


図4　アプリケーションから見た場合のデータのやりとり
専用の画像処理関数を呼ぶだけで，アプリケーションから回路を動かせる．アプリケーションから見た場合，回路で処理がなされているのか，パソコンで処理がなされているのか，区別はほとんどない．

● アプリケーションとAPIの間の交信をアレンジする

アプリケーションとAPIの間の交信については，今回とは別の方法もあります．

1）アプリケーションとハードウェアの処理を並行実行

FPGA側の処理時間が非常に長くかかるような場合，今回のようにアプリケーション実行とFPGA処理実行のどちらか一方しか動かさないのではなく，両方の処理実行を並列に

表4 APIから見た場合の制御回路メモリ・マップ（FPGAローカル）の例

アプリケーションから制御関数が呼ばれたとき，これらのレジスタを読み書きすることで処理が進む．アプリケーションからハードウェアを見た場合と異なり，ハードウェアの内部構成に密着することになり，ハードウェア（FPGA）とソフトウェア（パソコン）が並列に動いていることを強く意識する必要もある．なお，表の中のRは読み出し，Wは書き込みを意味する．

アドレス	セグメント	R/W	内　容	備　考
0x000000	0	RW	処理起動（書き込み時） 処理ステータス（読み出し時）	処理制御用レジスタ 回路の強制リセットも可能
0x002000	0	W	処理コマンド種類/処理バッファ指定	パラメータ用レジスタ
0x002004	0	W	画像横サイズ	パラメータ用レジスタ
0x002008	0	W	画像縦サイズ	パラメータ用レジスタ
0x002040-60	0	W	3×3フィルタ係数（計9個）	パラメータ用レジスタ
0x002400-2FFC	0	R	ヒストグラム計測値（各RGBにつき256階調）	パラメータ用レジスタ
0x003000-33FC	0	W	データ変換テーブル（各RGBにつき256階調）	パラメータ用レジスタ
0x000000-07FFFC	1	RW	画像メモリ・バッファ#0	データ用レジスタ
0x080000-0FFFFC	1	RW	画像メモリ・バッファ#1	データ用レジスタ
0x100000-17FFFC	1	RW	画像メモリ・バッファ#2	データ用レジスタ
0x180000-1FFFFC	1	RW	画像メモリ・バッファ#3	データ用レジスタ

行わせるのが得策です注5．そのような場合，処理終了を待たずに関数からリターンしておき，FPGA処理の実行終了通知をPCI経由の割り込みによって受け取るか，またはアプリケーションから処理終了を別途調べにいく形態にします．

アプリケーション・プログラムがハードウェア処理の終了をどのような方法で知るかは，プログラムの構造設計に大きく影響を与えるので，システム設計の初期段階でよく考えて決めるようにします．

2）ハードウェアからアプリケーションを起動させる

今回はアプリケーションがFPGAの処理を起動しますが，逆にFPGAがアプリケーションの処理を起動するケースも考えられます．例えば，FPGAボードが外部から画像データを受け取り，それをパソコンに渡すような場合です．この場合も，FPGAからパソコン側へ割り込みをかける形態になります．

● APIとハードの間はメモリ・マップを介してやりとり

次に，APIとハードウェア（図3のPCIドライバ）の間でどのようなやりとりが行われるかを見てみましょう．ここでは，FPGAのハードウェアとしての動きをかなり意識することになります．

APIとハードウェアの間では，割り込みを除き，メモリ空間の読み書きでデータをやり取りすることになります．このメモリ空間は，APIとペアになっている「画像処理IP

表5　API内部で使う補助関数群

表3の各API関数の内部では，これらの関数群を組み合わせて処理が行われる．これらの関数は，表4のFPGAローカル・メモリ空間への読み書きを行うものである．

補助関数群	内　容
piccopro_regif_init( )	回路を初期化し，各種情報（デバイス・ハンドラに相当）を取得する
piccopro_regif_setdata( )	処理するデータをFPGAボードへ転送する
piccopro_regif_getdata( )	処理するデータをFPGAボードから取り出す
piccopro_regif_setcmd( )	処理の種類をFPGAボードへ書き込む
piccopro_regif_setsize( )	画像のサイズをFPGAボードへ書き込む
piccopro_regif_setfilter( )	フィルタ係数をFPGAボードへ書き込む
piccopro_regif_settable( )	置換テーブルをFPGAボードへ書き込む
piccopro_regif_gethist( )	画像のヒストグラムをFPGAボードから読み出す
piccopro_regif_execcmd( )	設定した処理をFPGAに実行させる

注5：とくに，GUIを使うためにイベント・ドリブン・プログラミングを行う場合，そのように並列化を施していないと，ひどく応答性の悪いソフトウェアになってしまう．

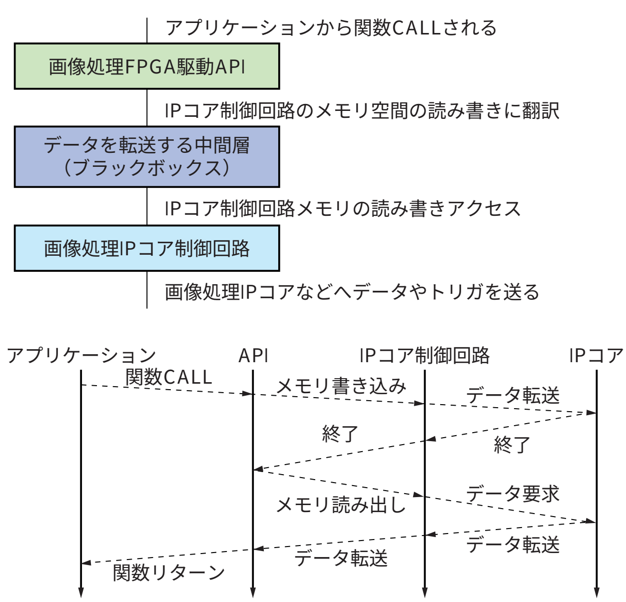


図5　APIから見た場合のデータのやりとり

FPGA側にメモリ・アドレス空間を持たせ，そこにデータを読み書きすることでAPIとFPGAが交信する．このメモリ空間はFPGAローカルのもので，パソコンのメモリ空間とは別である．なお，組み込みCPUを使う場合，回路のメモリ空間をCPUのメモリ空間に直接マップすることが多い．

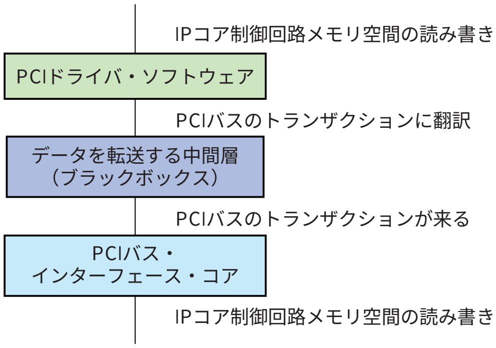


図6　PCIバスを使ったデータのやりとり
FPGAローカル・メモリ空間への読み書きは，さらにPCIバス上でのトランザクションに翻訳されて交信される．ここでは省略しているが，PCIドライバとPCIバス・インターフェース回路の間にもさまざまな処理階層が存在する．

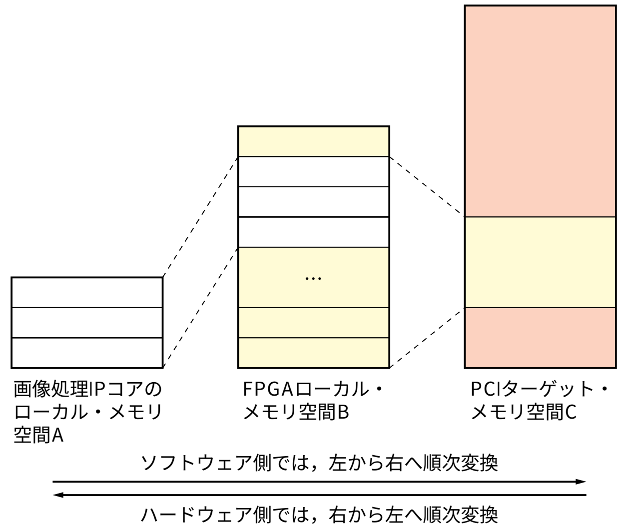


図7　メモリ空間のマッピングの繰り返し
表4のFPGAローカル・メモリ空間は，パソコンのPCIデバイスの固有メモリ空間の一部にマップされる．それはパソコンのメモリ空間の一部にもマップされる．また，画像処理IPコアそのものもローカルなメモリ空間を持っている．メモリ・アドレスは，ソフトウェア設計階層を上から下へ通過していく過程において図の左から右へ変換されていく．逆にハードウェア側では，図の右から左へ変換を行っていく．ソフトウェアやハードウェアの各階層の実装にあたっては，自分がどのアドレスを扱っているのかに注意を払う必要がある．

コア制御回路」の内部にローカルに作られているもので，パソコンやPCIのメモリ空間とは直接関係ありません．

表4に，実際のローカル・メモリ空間の例を示します．API関数はローカル・メモリ空間の読み書きを行うことで処理を進めます．例えば，**表3**におけるフィルタ処理実行関数piccopro_ipif_execfilterの動作は次のとおりです．参考文献(1)にも同じような例があるので参照してください．

1) パソコン上のメモリに置かれた画像データ(関数の引き数に渡される)を，ローカル・メモリ空間上のバッファへコピーする
2) 同じく，フィルタ係数値をローカル・メモリ空間上の該当アドレスへ書き込む
3) コマンド種類として「フィルタ処理」を指定する．これも該当アドレスへの書き込みで行う
4) 処理起動用コマンド・レジスタ(アドレス0x000000)に‘1’を書き込むと，FPGAが処理を開始する
5) ステータス・レジスタ(アドレス0x000000)が‘0’になるまで，つまり処理が終わるまでAPIが待つ

実際には，ローカル・メモリ空間の読み書きは**表5**のような補助関数群を使って行います．この補助関数群は，同じ処理を1ヵ所にまとめるために設けてあります．

以上の動作のようすを**図5**に示します．アプリケーションからAPI関数を呼び出した場合，APIではローカル・メモリ空間の読み書きを何度か行います．一方，APIとペアになっているIPコア制御回路は，ローカル・メモリ空間へのアクセス要求を受け取ると，IPコア本体とのデータ転送を適宜行います．

● PCIの階層でさらにアドレスの読み替えを行う

図6に示すように，ローカル・メモリ空間へのアクセスはPCIバスを中継して行われます．これは，ローカル・メモリ空間AをPCIターゲット・デバイス(ここではPCIバスに挿してあるFPGAボードのこと)のメモリ空間Bへマップし，その空間Bを読み書きすることで実現されます(**図7**)．メモリ空間Bは，さらにパソコンの実メモリ空間Cにマップされ，そこで初めてPCIバス上で現実のデータ

表6　PCIドライバ・ソフトウェアが提供する基本関数群(http://www.otto.to/~kasiwano/toppage11.htmから抜粋)
表5の各基本関数の内部では，これらの関数群を組み合わせて処理が行われる．PCIターゲット・デバイスへのデータの読み書きを行うものである．

基本関数群	内　容
_pciFindPciDevice( )	PCIターゲット・デバイスの各種情報(デバイス・ハンドラに相当)を取得する
_pciConfigReadShort( ), _pciConfigReadLong( )	PCIターゲット・デバイスのコンフィグレーション・レジスタの読み出し
_pciConfigWriteShort( )	同レジスタへの書き込み
_MemReadLong( )	PCIターゲット・デバイスのメモリ空間の読み出し
_MemWriteLong( )	同メモリ空間への書き込み
_MemReadBlockLong( )	同メモリ空間のバースト読み出し(高速)
_MemWriteBlockLong( )	同メモリ空間へのバースト書き込み(高速)

転送が行われます．

こうしたメモリ空間の読み替えは，ソフトウェアやハードウェアの各階層で行われます．メモリ空間AからBへの読み替えは**表5**の各関数内部で行われます．また，メモリ空間BからCへの読み替えは**表6**のPCIドライバ関数群で行われます．ハードウェア側では，CからBへ，BからAへと逆の変換が行われ，最終的に画像処理IPコアに到達します．

PCIドライバ関数群（**表6**）はPCIDEBUGという有名なものです．詳細は参考文献（3），（4）を参照してください．

● ソフトとハードの歩調を合わせる

API以下の階層を作るときには，ソフトウェアとハードウェアが同時に動いていると意識する必要があります．つまり，ソフトウェアとハードウェアがそれぞれバラバラに動いていてはデータのやりとりができないので，なんらかの方法で両者の交信タイミングを合わせる必要があるのです．これを「同期をとる」といいます．

今回は，次の2点によく気をつける必要があります．

1）ソフトウェア側は，画像処理IPコアの起動にあたって，起動可能かどうかを事前に確認する．なおかつコアを起動した後，その処理の終了を検知できるようにする

2）ハードウェア側はFPGAのローカル・メモリ空間への読み書きが発生したとき，そのアクセスが完了してからソフトウェア側へリターンする

これらの条件が守られないと，致命的な誤動作につながります．例えば，ハードウェア側がまだ処理中であるにもかかわらず，ソフトウェア側が「処理終了」と勘違いしてFPGAレジスタを読んだ場合，正しい演算結果は取り出せません．また，逆にハードウェアによる処理がすでに終了しているのに，ソフトウェア側が「まだ処理継続中」と勘違いすると，永久にソフトウェア処理を再開できなくなったりします（この状態を「デッドロック」と呼ぶ）．

上記**1）**については，ハードウェア側に「処理を開始できるかどうか」を表す専用のレジスタ，つまりステータス・レジスタを用意しておき，それをソフトウェア側がチェックすることで同期をとるようにします．ソフトウェアとハードウェアの速度差に注意する必要はありますが，基本的には処理開始前にはステータス・レジスタが'1'（処理開始可能）であることを，処理開始後にはステータス・レジスタが'1'→'0'→'1'となることを確認します（**図8**，**リスト1**）．

また，上記**2）**については，ハードウェア設計時にバス・アクセスにウェイトをかけるなどの方法で対処します．

同期をとる操作は，ソフトウェアとハードウェアの間だけでなく，ハードウェアとハードウェアの間でも用います．原則として，速度が異なる並列処理どうしを連携させるときには必要です．

リスト1　API内で使う補助関数の典型的なコーディング例

表4のレジスタへのデータ読み書きを，表6の関数を使い，PCIターゲット・デバイスへのデータ読み書きに翻訳する．並列に動いている回路側とソフトウェア側を同期させる（＝歩調を合わせる）ため，回路側の処理状況をチェックすることに注意が必要である．

FPGAに処理を行わせる

```
piccopro_regif_execcmd(piccoproinf *pi)
{
// IPコアのステータス・レジスタを読み，処理中でなくなるまで待つ
while ((_MemReadLong(pi->mem_base0 + 0) & 1) == 0)
  ‥中略‥
// IPコアのコマンド・レジスタへ1を書き，処理を開始させる
_MemWriteLong(pi->mem_base0 + 0, 1);
  ‥中略‥
// IPコアのステータス・レジスタを読み，処理中でなくなるまで待つ
while ((_MemReadLong(pi->mem_base0 + 0) & 1) == 0)
  ‥中略‥
}
```

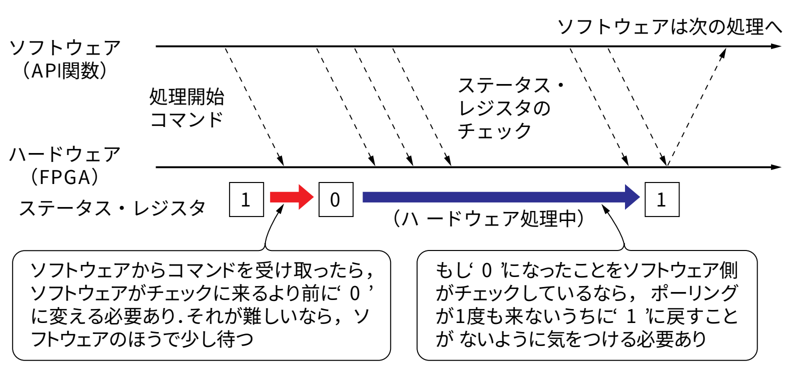


図8
ステータス・レジスタのポーリングによる，ソフトウェアとハードウェアの同期化（ハードウェアどうしでも同様）

ソフトウェアとハードウェアは実際には並列動作しているため，ソフトウェアが何か処理を起動したり，結果を取り出そうとするときは，ハードウェアが準備を完了していることをかならず確認しなければならない．これには，ハードウェア上にステータス・レジスタを設けてポーリング（繰り返しチェック）する方法がもっとも簡単である．ここでもっとも注意を要するのは，ステータス・レジスタの変化速度である．ハードウェアの速度とソフトウェアの速度が著しく異なる場合などに致命的な誤動作が発生することがある．

3 ハードウェアの詳細

ここまでは，ソフトウェアに重点を置いてシステムの構成を説明してきました．以下に，ハードウェア側がどのように作られているかを説明します．

● ブロック構成と全体的な動き

図9に，ハードウェア側のブロック図を示します．画像データを格納するSDRAMを除き，回路ブロックはFPGA内部に置かれます．各回路ブロックは図3のデータの流れどおりにつながっていますが，SDRAMの接続のしかたを少しだけ細工しています．このブロック図のもとで，回路がどのような動きをするかを見てみましょう．

1）画像処理を何も行っていないときの動き

FPGAがとくに画像処理を行っていないとき，SDRAMとのデータ交換や画像処理IPコアへの各種パラメータ設定をパソコン側から行うことができます．このときデータは，「パソコン→PCIバス・インターフェース→IPコア制御部→SDRAMまたはIPコア」という経路で流れます．

なお，表4のレジスタ群のうち，コマンド・レジスタやステータス・レジスタはIPコア制御部のほうに，それ以外のレジスタはIPコアのほうに置かれています．IPコア制御部では，コマンド・レジスタやステータス・レジスタへのアクセスは自分で処理しますが，それ以外のレジスタへのアクセスはそのままIPコアのほうへ中継します．

2）画像処理を起動するときの動き

パソコン側のソフトウェアが画像処理開始コマンド・レジスタ（表4のアドレス0x000000）へ‘1’を書き込むと，IPコア制御部から画像処理IPコアへ処理のスタート・トリガ（パルス）が送られます．それによって，画像処理IPコアが起動します．

3）画像処理中の動き

画像処理IPコアは，自分が処理を実行している間はSDRAMや内部レジスタへのアクセス権を握ります（図9のデータ・セレクタをIPコア側へ切り替える）．この間は，パソコン側からSDRAMなどへアクセスしようとしてもできなくなります．

また，画像処理IPコアはIPコア制御部へ，「処理を開始できるかどうか」を表すステータス信号をつねに送っています．IPコア制御部は，パソコンがステータス・レジスタを読みに来たら，そのステータス信号の値を返します．これによって，パソコン側は画像処理IPコアの状況をいつでも知ることができます．

4）画像処理後の動き

画像処理IPコアは，処理が終わるとSDRAMや内部レジスタへのアクセス権を手放します．その後，**1）**の状態へ戻ります．

● 3種類のクロック系統が必要

通常のRTL設計であれば，使うクロックは原則として1本しかありません．システムを構成する場合もそのように単一クロックですめば理想的なのですが，現実には複数クロックになる場合が少なくありません．そのおもな理由は，チップの外にある各種デバイスの動作クロックがまちまちであったり，チップの外にあるデバイスよりもチップ内部を高速動作させる必要があったりするからです．

本システムにおけるクロック系統図を図10に示します．今回はFPGAボードをPCIバスにつなぐので，PCIバス・インターフェース部はそのPCIバスから供給される33MHz

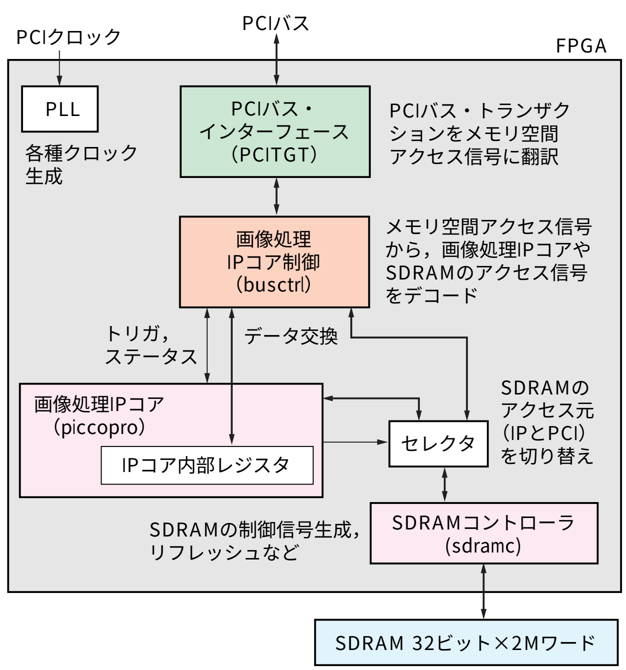


図9 ハードウェア側の全体構造

今回はデータの流れどおりに各コンポーネントを直接つなぐ方法をとっているが，もっと複雑な処理を行う場合は，CPUを用いて各IPコアをバスにぶら下げてもよい．PCIからのSDRAMアクセスは，画像処理IPコアの停止中に行える．

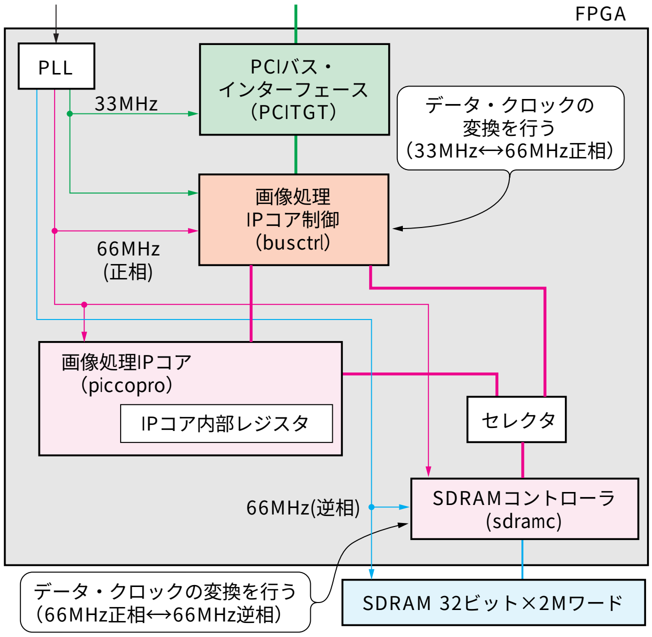


図10　クロック系統の構成

データの流れやコンポーネント間の接続を決める過程で，どこでどのようなクロックを使うかもおおむね定まってくる．とくに，PCIやSDRAMなど，外部回路との接続部ではクロック周波数が固定されることが多い．クロックだけでなくリセット・ラインの設計にも注意が必要．

クロックで動かすのが自然です[注6]．画像処理IPコアについては，33MHz動作では満足な性能が得られないため，もっと高速な66MHzクロックを使います．また，SDRAMチップについては，その信号の遅延設計のつごう上，SDRAMコントローラのクロックから多少位相をずらしたクロックをつながざるをえません．そのような理由から，図10に示した3種類のクロック系統が必要となっています．それらのクロックは，PCIのバス・クロックをもとにPLL（phase-locked loop）で作ります．

このように，どのようなクロックを用いるかは，かなりの部分がチップ外部の状況によって決まります．後述するように，各モジュールの論理設計に大きく影響するので，クロック系統はシステム設計の初期の段階で決定しておきます．

また，今回はとくに問題となりませんでしたが，クロック系統が複数ある場合，パワーONやリセット後のシステム起動がうまくいくかに十分注意する必要があります．例えば，パワーONしてからPLLが安定するまでは各システムを停止しておくのが安全です．また，システム・リセット信号についても，リセット解除時に一部の回路がほかより先に動き始めるような状況がどうしても発生するので，それで致命的な問題が起きないかどうかをシステム設計初期に十分検討する必要があります[注7]．

● 各ブロックの機能と交信プロトコル

それでは，図9の各ブロックについて説明していきます．

1) PCIバス・インターフェース

PCIバス・インターフェースとしては，参考文献(2)，(3)に収録されているPCIターゲット回路の設計データ(PCITGT)をVHDLからVerilog HDLに変換したうえで，おおむねそのまま用いています[注8]．PCITGTは，パソコンのPCIバスのプロトコル（バス信号の変化のしかた）を図11に示すような単純なメモリ・アクセス・プロトコルに変換する回路です．つまり，少々複雑に変化するPCIバスの信号を，普通のROMやSRAMなどとほぼ同じ「アドレス・バス＋データ・バス＋イネーブル信号」のインターフェースに変換します．PCIバスの詳細は参考文献(3)を参照してください．PCIバスのバースト転送，つまり複数アドレスへ連続的にデータ転送を行う操作も，1アドレスごとのメモリ・アクセスに切り分けられます．

2) 画像処理IPコア制御回路

画像処理IPコア制御回路は，PCIバスからのメモリ・アクセスの一部を処理し，後は画像処理IPコアへ中継します．この回路は，図11のようなインターフェースを持っており，PCIからのメモリ・サイクルがあると画像処理IPコアやSDRAMへのメモリ・サイクルを開始します．このように書くととても単純な回路ですみそうですが，実際にはやや複雑です．

まず，図7に示したとおり，PCIターゲット・デバイス上でのアドレスを表4のFPGAローカル・アドレスへ逆変換し，その結果に応じて画像処理IPコア，SDRAM，および自分自身（IPコア制御回路）のいずれにアクセスするべき

注6：「決して，異なるクロックを用いてはいけない」ということではないが，PCIバス・インターフェースの回路設計がとてもやっかいになる．
注7：各ブロックのリセットをどのような順番で解除していくかを検討し，場合によってはチップ外部からのリセット信号を受けて内部へ分配する（解除順も制御する）専用モジュールを設けることもある．
注8：回路製作にあたり，来栖川電工の井倉将実氏からPCIターゲット回路，SDRAMインターフェース回路，および関連情報などをご提供いただいた．本稿作成物（本誌Webサイトからダウンロード可能）には，来栖川電工の権利物も含まれる．本文ならびにソース・コード中に記載されているPCIベンダID：0x6809は来栖川電工固有のコードである．PCIデバイスID：0x4949は来栖川電工からCQ出版社に寄与されたデバイスIDである．これらのIDならびに本稿作成物の商業利用（ユーザ側で作成した機器への採用ならびに製品の販売）はいっさい認められていないので，注意していただきたい．

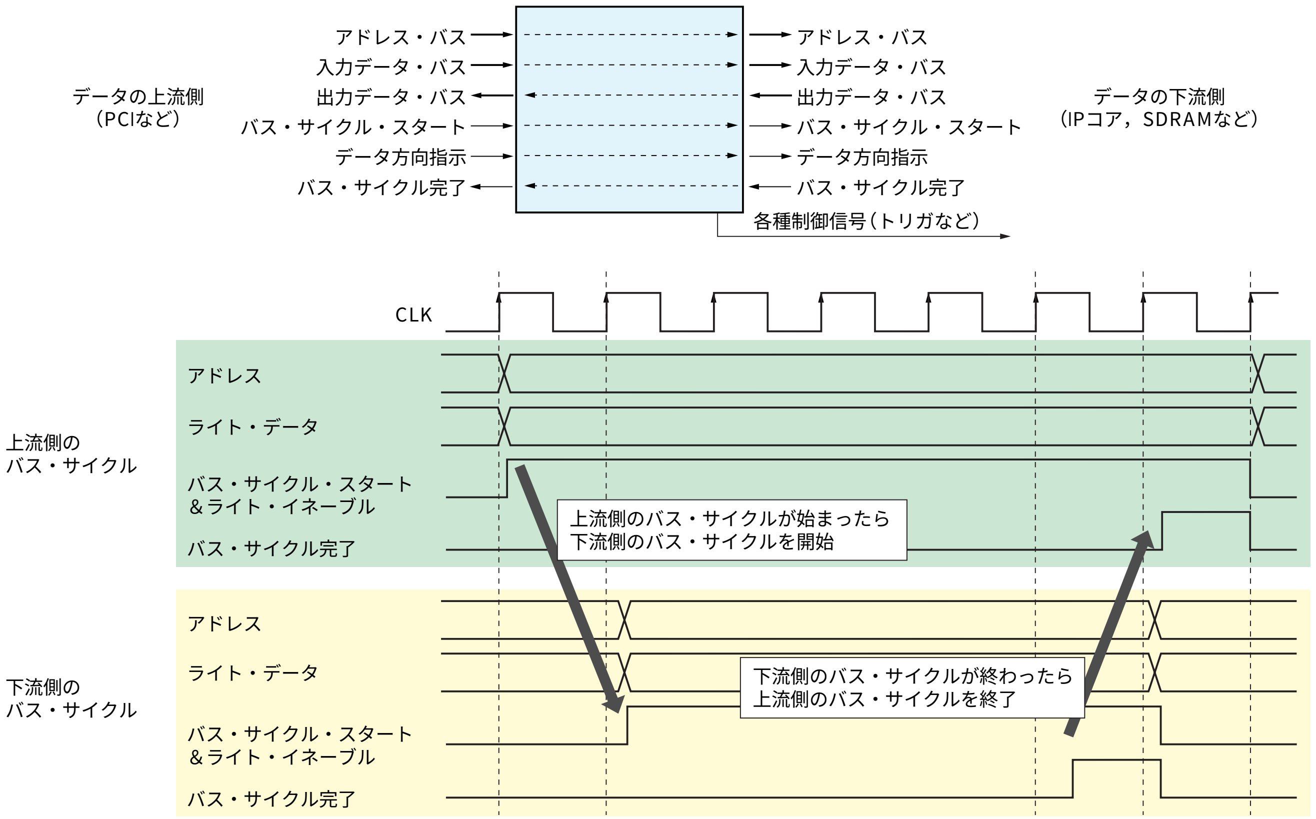


図11　各ブロックの典型的なI/O端子構成

各ブロックは，上流からのメモリ空間アクセスを解釈し，自分で処理すべきものは処理し，あるいはさらに下流のメモリ空間アクセスへ中継する．この図では上流と下流のバス端子やバス・プロトコルが同一であるが，一般には同じとは限らないので，それらの変換も適宜行う．とくに，クロック系が異なる場合は注意が必要である．上流，下流の接続に使う信号の種類は，データ・バス，アドレス・バス，バス・サイクルの開始・終了通知信号などが一般的．

リスト2
画像処理IPコア制御部の内部回路の典型例（抜粋）

PCIからFPGAメモリ空間へのライト・アクセスが来たとき，それをデコードしてIPコアのスタート・パルスを生成する部分を示す．リード・アクセスなども同じようにアドレスをデコードして処理する．

```
reg [2:0]       exec_start_reg;
wire            sel_command;
assign          sel_command  = (nPCI_CS0 == 1'b0 && PCI_A == 0) ? 1'b1 : 1'b0;
always @(posedge MEMCLK or negedge nRESET) begin
    if (nRESET == 1'b0) begin
        exec_start_reg <= 3'b000;
    end else begin
        if (sel_command == 1'b1 && nPCI_WE0 == 1'b0 && PCI_D_I[0] == 1'b1) begin
            exec_start_reg[0] <= exec_ready;
        end
        exec_start_reg[2:1] <= exec_start_reg[1:0];
    end
end
assign    exec_start = (~exec_start_reg[2]) & exec_start_reg[1];
```

PCIアドレスをデコード
コマンド・レジスタへ値 1 'を書き込む要求が来たら，画像処理IPコアのスタート・パルスを生成する
同期化
微分してスタート・パルスを生成

かを判断します．

次に，コマンド・レジスタやステータス・レジスタへのアクセスである場合，画像処理IPコアのスタート・パルス生成などを適宜行います（**リスト2**）．**リスト2**のコードはとくに難しいものではありませんが，参考文献（1）なども参照してください．

また，PCIバス側は33MHzクロック系ですが，画像処理IPコアやSDRAMは66MHzクロック系なので，クロック系の変換を行います．これは回路実装にあたって注意を要する部分なので，後で詳しく説明します．

3）SDRAMコントローラ

参考文献（2）に収録されている設計データ（sdramc）をVerilog HDLに変換したうえで，シミュレーション上の問題と異クロック系を接続するうえでの問題への対応策を施して使用しています．これは，**図11**のようなメモリ・アクセス信号を受けて，SDRAMの制御信号（RAS，CASな

ど）を生成し，SDRAMの読み書きを行う回路です．また，タイマを内蔵しており，一定時間ごとにSDRAMのリフレッシュ操作を行います．

4）画像処理IPコア

画像処理を行う本体です．本特集の第4章を参照してください．

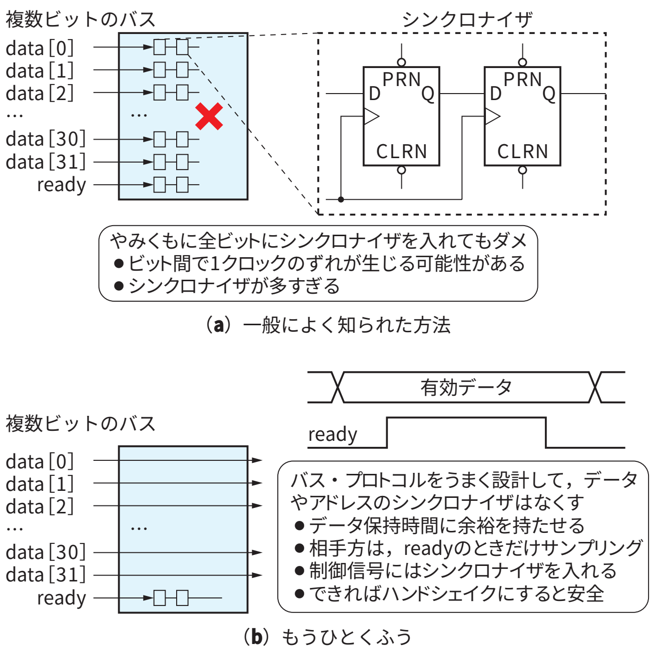


図12　クロックの異なるバスどうしの接続

クロック系の異なる回路どうしの接続にシンクロナイザ（1～2段のフリップフロップ）が必要なことはよく知られているが，実際にはそれだけでは不十分．バスなどの複数ビットのラインにおいて，ビット間で値のずれが生じうるからである．相手方でセットアップ/ホールド時間（フリップフロップでは，クロックの立ち上がり前後の一定時間，データ入力値を変化させてはならない．その時間のこと）の違反が起きないようにバス・プロトコルを設計すれば，データ・ラインやアドレス・ラインのシンクロナイザをなくすことができる．

● クロックの異なる回路どうしの接続にはひとくふう

図10に示したとおり，今回のシステムには異クロック系のブロックどうしの接続箇所が二つあります．一つは画像処理IPコア制御部（33MHz）と画像処理IPコア/SDRAMコントローラ（66MHz），もう一つはSDRAMコントローラとSDRAM（どちらも66MHzだが位相が異なる）です．

非常によく知られていることですが，このように異クロック系の信号どうしは，原則として直結してはいけません．回路中のフリップフロップ（FF）のセットアップ時間やホールド時間の違反が起きて回路が誤動作する場合があるからです．その初歩的な対策法は，信号をいったん1～2段のフリップフロップ（シンクロナイザと呼ぶ）で受けるというものです[(6)]．

ここでは，もう少し実用的な方法を二つ紹介します．まず，クロック系の異なるバス（複数ビット）どうしを接続する場合，そのアドレス・ラインやデータ・ラインの全ビットにシンクロナイザを入れるような設計を避けるようにします．各ビットの値がずれたり，シンクロナイザの数が多くなり過ぎたりするからです．そこで，図12のようにアドレス・ラインやデータ・ラインの変化タイミングに余裕を持たせ，信号の受け取り側でセットアップ/ホールド時間違反が起きないようなバス・プロトコルを使います．ただし，リクエストやアクノリッジといったバスの制御信号にはシンクロナイザが必要です．

また，1クロック幅のパルスを相手方へ送るような場合，相手のクロック周波数が自分のクロック周波数の2倍より遅ければ，図13のようにパルス幅を引き伸ばす必要があります．パルス幅は，相手のクロック周期の2倍以上にし

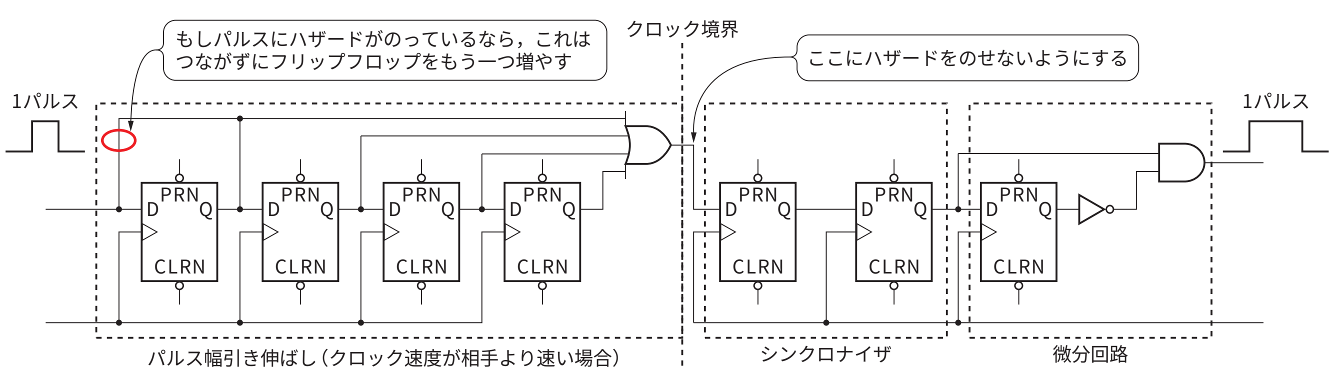


図13　クロック速度が異なる場合のパルスの受け渡し

クロック系の異なる回路へ処理スタート信号や処理完了信号を送るときは，1クロック・パルスは避けてハンドシェイク（1回データをやりとりするたびに，かならず送り手と受け手で確認を取り合うプロトコル．詳しくは参考文献（1），（6）を参照）を行うとよい．しかし，既存のIPコアを利用する場合など，どうしても1パルスの信号しか得られないならば，図のように相手が受け取れるだけのパルス長（相手のクロック周期の2倍以上）へ引き伸ばす必要がある．また，相手方へ送る信号にハザードをのせないことも重要．ダイナミック・ハザード（信号値が確定するまでの信号のバタバタ）がパルスとして伝播してしまう可能性がある．

ます．そのようにしないと，相手方がパルスを取りこぼす場合があるからです．今回は，画像処理IPコア制御部とSDRAMコントローラの間の接続にこの方法を利用しています．

以上のような対策をいい加減に済ませると，「ハードウェアの実機がときどき誤動作したり，デッドロックする」というタチの悪いバグが発生しやすくなります．

4 実機を動かすためのテスト戦略

では，最後に実機を動かすうえでのテストのやりかたについて説明します．

● 経験則から見る実機が動かない理由と原因

以下に，実機で回路をテストするうえでの筆者の経験則を紹介します．

- **実機が動かないときはケアレス・ミス** ── 実機が動かないときの原因はほんの小さなケアレス・ミスである場合がもっとも多い．例えば，ちょっとしたタイミングの勘違いや，正論理と負論理の勘違いなどである．アルゴリズムがおかしいようなことはどちらかと言えばまれ．
- **誤動作と原因にはギャップがある** ── 表立って見える誤動作とほんとうの原因の間のギャップが大きく，なかなか原因をつかめないことが多い．原因がわかった後でも，なぜそのような誤動作につながったのか説明のつかないことがしばしばある[注9]．
- **チップ外部や異クロック系との接続が問題になりやすい** ── チップ外の回路や異クロック系の接続部では問題が発生しやすい．まったく動かない場合もあるし，ときどき動かなくなるというように不安定な場合もある[注10]．チップ外にあるバスやデバイスなどで，その仕様書と実機の動きが違っている場合もときどきある．こうした問題は，シミュレーションではとても発見しにくい．
- **同期回路の不ぐあいではクロックや遅延をチェック** ── チップ内の論理部分，とくに単一クロックの同期回路になっている部分は，まずまちがいなくシミュレーションどおりに動く．同期回路にもかかわらず動かない場合，クロック周期が速すぎるか，回路入出力部で遅延が間に合っていないことが多い．

これらを念頭に置いて，以下を読み進めてください．

● まずシミュレーションを行ってから実機テストへ移る

実機を動かす前に，めんどうでもシミュレーションをひととおり通すことをお勧めします．今回の事例であれば，PCIバスやSDRAMコントローラなどについて，いきなり実機でデバッグしながら回路を作り込むことも不可能ではありません．しかし，一見回り道のようでも，シミュレーションを行うことには，次のようなメリットがあります．

- 実機ではわかりづらいケアレス・ミスの大半をすぐ除去できる
- もし実機が動かなかった場合，シミュレーションしてある部分を原因から除外できる
- 複数人で分担して回路を作成する場合，同じシミュレーション環境を共有しておけば回路結合時のトラブルが減る

実機による合わせ込みを行う箇所を，遅延タイミングの微調整くらいにしておくと，結果的に回路設計がずっと効率的に進みます．今回は，SDRAMチップやPCIイニシエータ（パソコン側のPCIインターフェースに相当）など

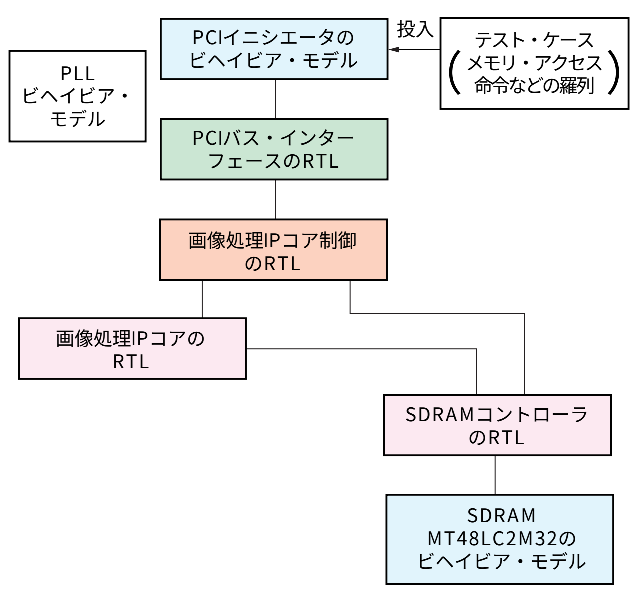


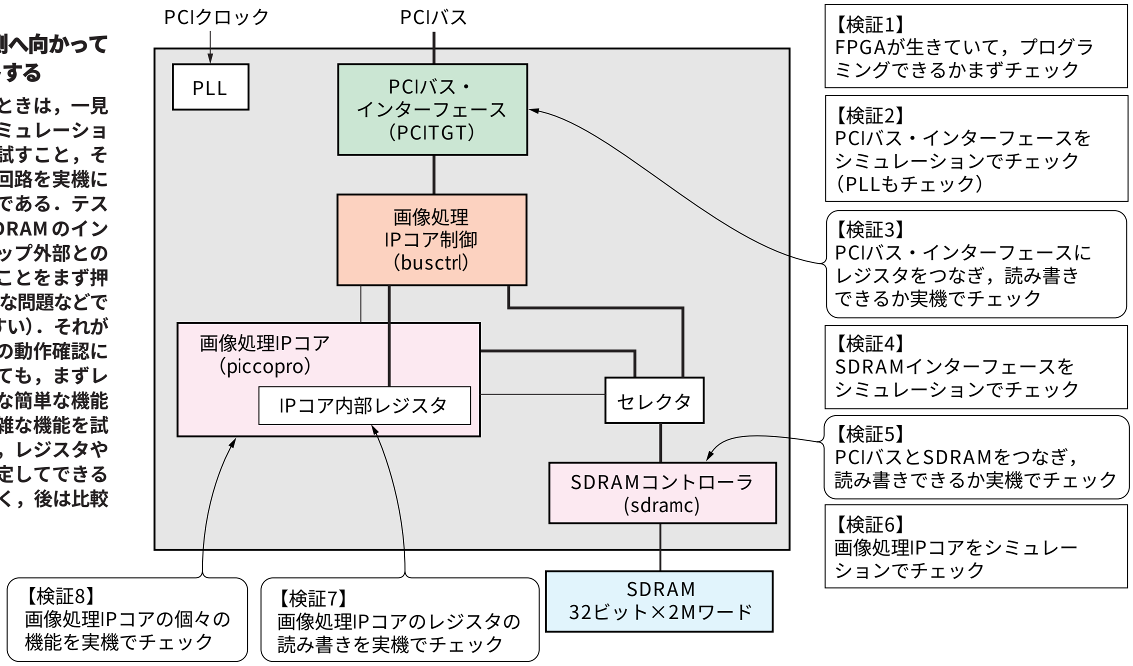
図14　今回構築したハードウェアのシミュレーション環境
今回はソフトウェアとハードウェアの協調シミュレーションまでは行っていないが，PCI経由でメモリ・アクセス命令を送り込むテストは実施している．PCIやSDRAMのビヘイビア・モデルは無償で入手できる．きっちりしたシミュレーション環境を構築しておけば，複数人での分担開発もスムーズになる．

注9：例えば，ソフトウェア・プログラミングのメモリ・リークに起因する誤動作では，なぜそのような誤動作に至ったのかを説明しにくい．原因と誤動作の間に多くの処理がはさまりがちだからである．感覚としてはそれと同じ．

注10：まったく動かない場合はケアレス・ミス，動きが不安定な場合はタイミング（遅延）設計や異クロック系どうしの接続の問題をまず疑ってみるとよい．

3

図15
チップの外側から内側へ向かってボトムアップにテストする
実機を動かそうとするときは，一見回り道のようだが，シミュレーションを行ってから実機で試すこと，そしていきなりすべての回路を実機に実装しないことが得策である．テストでは，PCIバスやSDRAMのインターフェースなど，チップ外部との接続部が安定して動くことをまず押さえる（電気的，機械的な問題などで不安定な動作が起きやすい）．それができて初めて内部論理の動作確認に移る．内部論理についても，まずレジスタ読み書きのような簡単な機能から始めて，徐々に複雑な機能を試していく．経験的には，レジスタやメモリの読み書きが安定してできるまでが（心理的にも）長く，後は比較的早い．



のチップ外の部品については，参考文献(7)～(9)などから無償のビヘイビア・モデルを入手し，PLLについてはQuartus II（Altera社のFPGA開発ツール）にビヘイビア・モデルを生成させることで，**図14**のシミュレーション環境を作りました．

● 実チップは外側から内側へ向かってテストする

シミュレーションで動作を確認したら，実機によるテストを始めます．このとき，いきなりFPGAにすべての回路を実装して動かそうとしたりせず，**図15**に示すように足回りの部分から順番に固めていきます．

まず，そのボードが初めて使うものであれば，電源が入るか，FPGAデバイスが生きているか，FPGAにプログラミング（コンフィグレーション）できるか，クロック信号が入っているかなどについて簡単な回路[注11]でテストしておきます．

それからターゲットの回路を動作させていくことになりますが，まずPCIバス・インターフェースやSDRAMコントローラなど，チップ外と交信するブロックだけを個別に動かしてテストします．このとき，データ値にかかわらず正しくデータ転送が行えるか，ときどきデータが化けて転送されることがないか[注12]，メモリのリフレッシュが行われているか，というような安定性をとくに重点的にチェックします．こうした基本動作が確実に行えないと，残りの回路のデバッグ時に問題の切り分けができなくなるからです．

ここまで到達して初めて，画像処理IPコアなど，チップ内部の論理をFPGAに実装することができます．IPコアのテストでは，まずレジスタの読み書きがチップ外から実行できるかどうかを調べてください．レジスタの読み書きさえできれば，後はシミュレーションどおりにすんなり動く場合が多いようです．

● ソフト・ハードのペアは中央から外側へ向かってテスト

今回のようにソフトウェアとハードウェアの組み合わさったシステムの場合，これまで説明した実機テスト手順の中で，ソフトウェアもあわせてテストしていくことになります．そのとき，**図3**に示したペアの真ん中から外側へ向かってテストは進みます（**図16**）．

今回の場合，FPGA上のPCIバス・インターフェース回路をテストする際に，ソフトウェア側のPCIドライバ（PCIDEBUG）もあわせてテストします．回路のテストがチップの外側から内部へ進んでいくにしたがって，ソフトウェアもPCIドライバ→FPGA駆動API→アプリケーションという順番で動かしていきます．

注11：例えば，スイッチを押すとLEDが点滅するような回路などがよい．
注12：データとしてはオール‘0’，オール‘1’，1ビットだけ‘1’，1ビットだけ‘0’，2進の“0101…0101”や“1010…1010”などの値を使うと，クロストークなどの電気的要因による誤動作や結線ミスをチェックできる．

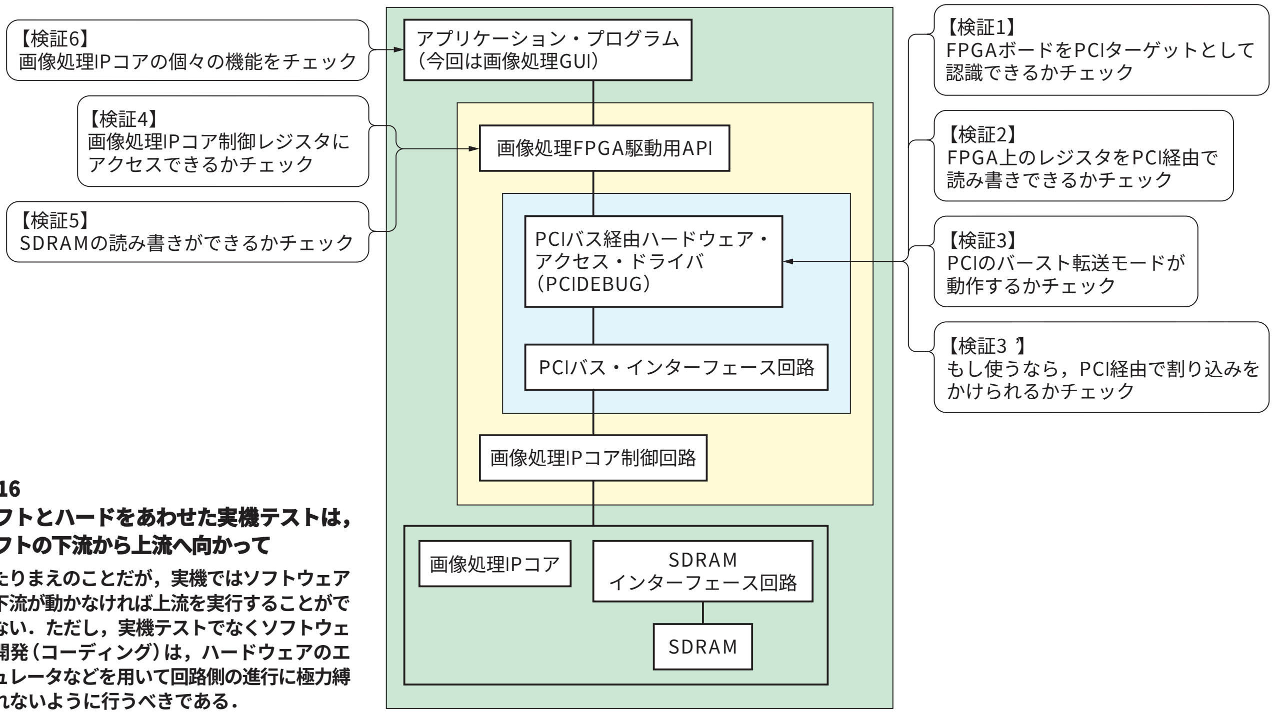


図16
ソフトとハードをあわせた実機テストは，ソフトの下流から上流へ向かって
あたりまえのことだが，実機ではソフトウェアの下流が動かなければ上流を実行することができない．ただし，実機テストでなくソフトウェア開発（コーディング）は，ハードウェアのエミュレータなどを用いて回路側の進行に極力縛られないように行うべきである．

なお，ここで述べているのは「実機テストにおいてはソフトウェアを下流（ハードウェア側）から上流へ向かって動かしていく」という意味です．ソフトウェア作成そのものを下から上へ行うという意味ではないので注意してください．実務においては，ハードウェアの実機ができる前にソフトウェアを先行開発することがごく普通に行われています．

参考・引用*文献

(1) S.Mori；演算回路に不可欠なインターフェースの設計，Design Wave Magazine，2005年3月号，pp.56-69（本稿よりもさらにシンプルな例題を用いた説明）．
(2) Stratix評価ボードに関するCQ出版社Webサイト；http://www.cqpub.co.jp/eda/Stratix/default.htm
(3) Interface編集部編；改定新版 PCIデバイス設計入門，CQ出版社，2005年4月（今回用いたPCIインターフェース回路やソフトウェア・ライブラリPCIDEBUGなどを収録）．
(4) PCIDEBUGに関するWebサイト；http://www.otto.to/~kasiwano/newpage16.htm（PCI経由でハードウェアにアクセスするためのソフトウェア・ライブラリ）．
(5) PCIコンフィグレジスター情報；http://www.vector.co.jp/soft/winnt/hardware/se303974.html（PCIターゲット・デバイスのメモリ/レジスタの読み書きツール）．
(6) 森岡澄夫；HDLによる高性能ディジタル回路設計，CQ出版社，2002年11月（回路設計のいろいろなテクニック）．
(7) トライゲートのWebサイト；http://www.trigate.co.jp/Download_TOP.htm（PCIイニシエータのシミュレーション・モデル，無償，筆者はVHDL版を利用）．
(8) ラティス・セミコンダクターのWebサイト（PCIターゲット32ビット/33MHz）；http://www.latticesemi.co.jp/products/intellectualproperty/referencedesigns/pcitarget32bit33mhz.html（PCIターゲット・コア，無償）．
(9) 米国Micron Technology社のWebサイト（SDRAM Component Part List）；http://www.micron.com/products/dram/sdram/part.aspx?part=MT48LC2M32B2P-6#SIM（今回使ったFPGAボード上のSDRAM“MT48LC2M32B2P”のシミュレーション・モデル）．

もりおか・すみお
NEC システムデバイス研究所

<筆者プロフィール>
森岡 澄夫．NTT，IBM，ソニーの各社で高性能IPコアやシステムLSIの研究開発を行ってきた．PSP搭載セキュリティ・システム回路などの開発によりSony MVP2004．現在は，多くの実設計経験をベースに，NEC システムデバイス研究所にて動作合成システム「Cyber」の研究開発に従事．趣味は電子工作（年がばれる？），子育て，ロケット打ち上げ見物．http://www02.so-net.ne.jp/~morioka